Telegram Open Network Blockchain

Nikolai Durov

September 5, 2018

Abstract

The aim of this text is to provide a detailed description of the Telegram Open Network (TON) Blockchain.

Introduction

This document provides a detailed description of the TON Blockchain, including its precise block format, validity conditions, TON Virtual Machine (TVM) invocation details, smart-contract creation process, and cryptographic signatures. In this respect it is a continuation of the TON whitepaper (cf. [3]), so we freely use the terminology introduced in that document.

Chapter **1** provides a general overview of the TON Blockchain and its design principles, with particular attention to the introduction of compatibility and validity conditions and the implementation of message delivery guarantees. More detailed information, such as the TL-B schemes that describe the serialization of all required data structures into trees or collections ("bags") of cells, is provided in subsequent chapters, culminating in a complete description of the TON Blockchain (shardchain and masterchain) block layout in Chapter **5**.

A detailed description of the elliptic curve cryptography used for signing blocks and messages, also accessible through TVM primitives, is provided in Appendix **A**. TVM itself is described in a separate document (cf. [4]).

Some subjects have intentionally been left out of this document. One is the Byzantine Fault Tolerant (BFT) protocol used by the validators to determine the next block of the masterchain or a shardchain; that subject is left for a forthcoming document dedicated to the TON Network. And although

this document describes the precise format of TON Blockchain blocks, and discusses the blockchain's validity conditions and serialized invalidity proofs,[1] it provides no details about the network protocols used to propagate these blocks, block candidates, collated blocks, and invalidity proofs.

Similarly, this document does not provide the complete source code of the masterchain smart contracts used to elect the validators, change the configurable parameters or get their current values, or punish the validators for their misbehavior, even though these smart contracts form an important part of the total blockchain state and of the masterchain block zero. Instead, this document describes the location of these smart contracts and their formal interfaces.[2] The source code of these smart contracts will be provided separately as downloadable files with comments.

Please note that the current version of this document describes a preliminary test version of the TON Blockchain; some minor details are likely to change prior to launch during the development, testing, and deployment phases.

[1] As of August 2018, this document does not include a detailed description of serialized invalidity proofs, because they are likely to change significantly during the development of the validator software. Only the general design principles for consistency conditions and serialized invalidity proofs are discussed.

[2] This is not included in the present version of this document, but will be provided in a separate appendix to a future revision.

Contents

1 Overview

This chapter provides an overview of the main features and design principles of the TON Blockchain. More detail on each topic is provided in subsequent chapters.

1.1 Everything is a bag of cells

All data in the blocks and state of the TON Blockchain is represented as a collection of *cells* (cf. [3, 2.5]). Therefore, this chapter begins with a general discussion of cells.

1.1.1. TVM cells. Recall that the TON Blockchain, as well as the TON Virtual Machine (TVM; cf. [4]), represents all permanently stored data as a *collection* or *bag* of so-called *cells*. Each cell consists of up to 1023 data bits and up to four references to other cells. Cyclic cell references are not allowed, so the cells are usually organized into *trees of cells*, or rather *directed acyclic graphs (DAGs) of cells*.[3] Any value of an abstract algebraic (dependent) data type may be represented (serialized) as a tree of cells. The precise way of representing values of an abstract data type as a tree of cells is expressed by means of a *TL-B scheme*.[4] A more thorough discussion of different kinds of cells may be found in [4, 3.1].

1.1.2. Application to TON Blockchain blocks and state. The above is particularly applicable to the blocks and state of the TON Blockchain, which also are values of certain (quite convoluted) dependent algebraic data types. Therefore, they are serialized according to various TL-B schemes (which are gradually presented throughout this document), and are represented as a collection or bag of cells.

1.1.3. The layout of a single cell. Each single cell consists of up to 1023 data bits and up to four references to other cells. When a cell is kept in memory, its exact representation is implementation-dependent. However,

[3] Completely identical cells are often identified in memory and in disk storage; this is the reason why trees of cells are transparently transformed into DAGs of cells. From this perspective, a DAG is just a storage optimization of the underlying tree of cells, irrelevant for most considerations.

[4] Cf. [4, 3.3.3–4], where an example is given and explained, pending a more complete reference

there is a standard representation of cells, useful, for instance, for serializing cells for file storage or network transmission. This "standard representation" or "standard layout" CellRepr(c) of a cell c consists of the following:

- Two *descriptor bytes* come first, sometimes denoted by d_1 and d_2. The first of these bytes d_1 equals (in the simplest case) the number of references $0 \le r \le 4$ in the cell. The second descriptor byte d_2 encodes the bit length l of the data part of the cell as follows: the first seven bits of d_2 equal $\lfloor l/8 \rfloor$, the number of complete data bytes present in the cell, while the last bit of d_2 is the *completion tag*, equal to one if l is not divisible by eight. Therefore,

$$d_2 = 2\lfloor l/8 \rfloor + [l \bmod 8 \neq 0] = \lfloor l/8 \rfloor + \lceil l/8 \rceil \tag{1}$$

  where $[A]$ equals one when condition A is true, and zero otherwise.

- Next, $\lceil l/8 \rceil$ data bytes follow. This means that the l data bits of the cell are split into groups of eight, and each group is interpreted as a big-endian 8-bit integer and stored into a byte. If l is not divisible by eight, a single binary one and a suitable number of binary zeroes (up to six) are appended to the data bits, and the completion tag (the least significant bit of the descriptor byte d_2) is set.

- Finally, r references to other cells follow. Each reference is normally represented by 32 bytes containing the SHA256 hash of the referenced cell, computed as explained below in **1.1.4**.

In this way, the standard representation CellRepr(c) of a cell c with l data bits and r references is $2 + \lfloor l/8 \rfloor + \lceil l/8 \rceil + 32r$ bytes long.

1.1.4. The SHA256 hash of a cell. The SHA256 hash of a cell c is recursively defined as the SHA256 of the standard representation CellRepr(c) of the cell in question:

$$\text{Hash}(c) := \text{SHA256}(c) := \text{SHA256}\big(\text{CellRepr}(c)\big) \tag{2}$$

Because cyclic cell references are not allowed (the relationships among all cells must constitute a directed acyclic graph, or DAG), the SHA256 hash of a cell is always well-defined.

Furthermore, because SHA256 is tacitly assumed to be collision-resistant, we assume that all the cells that we encounter are completely determined

by their hashes. In particular, the cell references of a cell c are completely determined by the hashes of the referenced cells, contained in the standard representation CellRepr(c).

1.1.5. Exotic cells. Apart from the *ordinary* cells (also called *simple* or *data* cells) considered so far, cells of other types, called *exotic cells*, sometimes appear in the actual representations of TON Blockchain blocks and other data structures. Their representation is somewhat different; they are distinguished by having the first descriptor byte $d_1 \geq 5$ (cf. [4, 3.1]).

1.1.6. External reference cells. *(External) reference cells*, which contain the 32-byte sha256(c) of a "true" data cell c instead of the data cell itself, are one example of exotic cells. These cells can be used in the serialization of a bag of cells corresponding to a TON Blockchain block in order to refer to data cells absent in the serialization of the block itself, but assumed to be present somewhere else (e.g., in the previous state of the blockchain).

1.1.7. Transparency of reference cells with respect to most operations. Most cell operations do not observe any reference cells or other "exotic" kinds of cells; they see only data cells, with any reference cell transparently replaced by the cell referred to. For example, when the *transparent* cell hash Hash$^\flat(c)$ is recursively computed, the hash of a reference cell is set to be equal to the hash of the cell referred to, not the hash of the standard representation of the reference cell.

1.1.8. Transparent hash and representation hash of a cell. In this way, sha256$^\flat(c)$ = Hash$^\flat(c)$ is the *transparent hash* of a cell c (or the tree of cells rooted in c).

However, sometimes we need to reason about the exact representation of a tree of cells present in a block. To this end, a *representation hash* Hash$^\sharp(c)$ is defined, which is not transparent with respect to reference cells and other exotic types of cells. We often say that the representation hash of c is "the" hash of c, because it is the most oftenly used hash of a cell.

1.1.9. Use of representation hashes for signatures. Signatures are an excellent example of the application of representation hashes. For instance:

- Validators sign the representation hash of a block, not just its transparent hash, because they need to certify that the block does contain the required data, not just some external references to them.

- When external messages are signed and sent by off-chain parties (e.g., human clients using an application to initiate blockchain transactions), if external references may be present in some of these messages, it is the representation hashes of the messages that must be signed.

1.1.10. Higher hashes of a cell. In addition to the transparent and representation hashes of a cell c, a sequence of *higher hashes* $\textsc{Hash}_i(c)$, $i = 1, 2, \ldots$ may be defined, which eventually stabilizes at $\textsc{Hash}_\infty(c)$. (More detail may be found in [4, 3.1].)

1.2 Principal components of a block and the blockchain state

This section briefly describes the principal components of a block and of the blockchain state, without delving too much into the details.

1.2.1. The Infinite Sharding Paradigm (ISP) applied to blockchain block and state. Recall that according to the Infinite Sharding Paradigm, each account can be considered as lying in its separate "accountchain", and the (virtual) blocks of these accountchains are then grouped into shardchain blocks for efficiency purposes. Specifically, the state of a shardchain consists, roughly speaking, of the states of all its "accountchains" (i.e., of all accounts assigned to it); similarly, a block of a shardchain essentially consists of a collection of virtual "blocks" for some accounts assigned to the shardchain.[5]

We can summarize this as follows:

$$\mathit{ShardState} \approx \mathit{Hashmap}(n, \mathit{AccountState}) \tag{3}$$

$$\mathit{ShardBlock} \approx \mathit{Hashmap}(n, \mathit{AccountBlock}) \tag{4}$$

where n is the bit length of the $\mathit{account\_id}$, and $\mathit{Hashmap}(n, X)$ describes a partial map $\mathbf{2}^n \dashrightarrow X$ from bitstrings of length n into values of type X.

Recall that each shardchain—or, more precisely, each shardchain block[6]—corresponds to all accountchains that belong to the same "workchain" (i.e., have the same $\mathit{workchain\_id} = w$) and have an $\mathit{account\_id}$ beginning with

[5] If there are no transactions related to an account, the corresponding virtual block is empty and is omitted in the shardchain block

[6] Recall that TON Blockchain supports *dynamic* sharding, so the shard configuration may change from block to block because of shard merge and split events. Therefore, we cannot simply say that each shardchain corresponds to a fixed set of accountchains.

the same binary prefix s, so that (w, s) completely determines a shard. Therefore, the above hashmaps must contain only keys beginning with prefix s.

We will see in a moment that the above description is only an approximation: the state and block of the shardchain need to contain some extra data that are not split according to the *account_id* as suggested by (3).

1.2.2. Split and non-split part of the shardchain block and state. A shardchain block and its state may each be classified into two distinct parts. The parts with the ISP-dictated form of (3) will be called the *split* parts of the block and its state, while the remainder will be called the *non-split* parts.

1.2.3. Interaction with other blocks and the outside world. Global and local consistency conditions. The non-split parts of the shardchain block and its state are mostly related to the interaction of this block with some other "neighboring" blocks. The global consistency conditions of the blockchain as a whole are reduced to internal consistency conditions of separate blocks by themselves as well as external local consistency conditions between certain blocks (cf. **1.3**).

Most of these local consistency conditions are related to message forwarding between different shardchains, transactions involving more than one shardchain, and message delivery guarantees. However, another group of local consistency conditions relates a block with its immediate antecessors and successors inside a shardchain; for instance, the initial state of a block usually must coincide with the final state of its immediate antecessor.[7]

1.2.4. Inbound and outbound messages of a block. The most important components of the non-split part of a shardchain block are the following:

- *InMsgDescr* — The description of all messages "imported" into this block (i.e., either processed by a transaction included in the block, or forwarded to an output queue, in the case of a transit message travelling along the path dictated by Hypercube Routing).
- *OutMsgDescr* — The description of all messages "exported" or "generated" by the block (i.e., either messages generated by a transaction included in the block, or transit messages with destination not belonging to the current shardchain, forwarded from *InMsgDescr*).

[7]This condition applies if there is exactly one immediate antecessor (i.e., if a shardchain merge event did not occur immediately before the block in question); otherwise, this condition becomes more convoluted.

1.2.5. Block header. Another non-split component of a shardchain block is the *block header*, which contains general information such as (w, s) (i.e., the *workchain_id* and the common binary prefix of all *account_id*s assigned to the current shardchain), the block's *sequence number* (defined to be the smallest non-negative integer larger than the sequence numbers of its predecessors), *logical time*, and *generation unixtime*. It also contains the hash of the immediate antecessor of the block (or of its two immediate antecessors in the case of a preceding shardchain merge event), the hashes of its initial and final states (i.e., of the states of the shardchain immediately before and immediately after processing the current block), and the hash of the most recent masterchain block known when the shardchain block was generated.

1.2.6. Validator signatures, signed and unsigned blocks. The block described so far is an *unsigned block*; it is generated in its entirety and considered as a whole by the validators. When the validators ultimately sign it, the *signed block* is created, consisting of the unsigned block along with a list of validator signatures (of a certain representation hash of the unsigned block, cf. **1.1.9**). This list of signatures is also a non-split component of the (signed) block; however, since it lies outside the unsigned block, it is somewhat different from the other data kept in a block.

1.2.7. Outbound message queue of a shardchain. Similarly, the most important non-split part of the shardchain state is *OutMsgQueue*, the outbound message queue. It contains *undelivered* messages included into *OutMsgDescr*, either by the last shardchain block leading to this state or by one of its antecessors.

Originally, each outbound message is included into *OutMsgQueue*; it is removed from the queue only after it has either been included into the *InMsgDescr* of a block of a "neighboring" shardchain (the next one with respect to Hypercube Routing), or has been delivered to (i.e., has appeared in the *InMsgDescr* of) its ultimate destination shardchain via Instant Hypercube Routing. In both cases, the *reason* for the removal of a message from the *OutMsgQueue* is made explicit in the *OutMsgDescr* of the block in which such a state transformation has occurred.

1.2.8. Layout of *InMsgDescr*, *OutMsgDescr* and *OutMsgQueue*. All of the most important non-split shardchain data structures related to messages are organized as *hashmaps* or *dictionaries* (implemented by means of Patricia trees serialized into a tree of cells as described in [4, 3.3]), with the

following keys:

- The inbound message description *InMsgDescr* uses the 256-bit message hash as a key.
- The outbound message description *OutMsgDescr* uses the 256-bit message hash as a key.
- The outbound message queue *OutMsgQueue* uses the 352-bit concatenation of the 32-bit destination *workchain_id*, the first 64 bits of destination address *account_id*, and the 256-bit message hash as a key.

1.2.9. The split part of the block: transaction chains. The split part of a shardchain block consists of a hashmap mapping some of the accounts assigned to the shardchain to "virtual accountchain blocks" *AccountBlock*, cf. (3). Such a virtual accountchain block consists of a sequential list of *transactions* related to that account.

1.2.10. Transaction description. Each transaction is described in the block by an instance of the *Transaction* type, which contains in particular the following information:

- A reference to exactly one *inbound message* (which must be present in *InMsgDescr* as well) that has been *processed* by the transaction.
- References to several (maybe zero) *outbound messages* (also present in *OutMsgDescr* and most likely included in *OutMsgQueue*) that have been *generated* by the transaction.

The transaction consists of an invocation of TVM (cf. [4]) with the code of the smart contract corresponding to the account in question loaded into the virtual machine, and with the data root cell of the smart contract loaded into the virtual machine's register `c4`. The inbound message itself is passed in the stack as an argument to the smart contract's `main()` function, along with some other important data, such as the amount of TON Grams and other defined currencies attached to the message, the sender account address, the current balance of the smart contract, and so on.

In addition to the information listed above, a *Transaction* instance also contains the original and final states of the account (i.e., of the smart contract), as well as some of the TVM running statistics (gas consumed, gas price, instructions performed, cells created/destroyed, virtual machine termination code, etc.).

1.2.11. The split part of the shardchain state: account states. Recall that, according to (3), the split part of the shardchain state consists of a hashmap mapping each "defined" account identifier (belonging to the shardchain in question) to the *state* of the corresponding account, given by an instance of the *AccountState* type.

1.2.12. Account state. The account state itself approximately consists of the following data:

- Its *balance* in Grams and (optionally) in some other defined cryptocurrencies/tokens.
- The *smart-contract code*, or the hash of the smart-contract code if it will be provided (uploaded) later by a separate message.
- The persistent *smart-contract data*, which can be empty for simple smart contracts. It is a tree of cells, the root of which is loaded into register `c4` during smart-contract execution.
- Its *storage usage statistics*, including the number of cells and bytes kept in the persistent storage of the smart contract (i.e., inside the blockchain state) and the last time a storage usage payment was exacted from this account.
- An optional *formal interface description* (intended for smart contracts) and/or *user public information* (intended mostly for human users and organizations).

Notice that there is no distinction between "smart contract" and "account" in the TON Blockchain. Instead, "simple" or "wallet" accounts, typically employed by human users and their cryptocurrency wallet applications for simple cryptocurrency transfers, are just simple smart contracts with standard (shared) code and with persistent data consisting of the public key of the wallet (or several public keys in the case of a multi-signature wallet; cf. **1.7.6** for more detail).

1.2.13. Masterchain blocks. In addition to shardchain blocks and their states, the TON Blockchain contains *masterchain blocks* and the *masterchain state* (also called the *global state*). The masterchain blocks and state are quite similar to the shardchain blocks and state considered so far, with some notable differences:

- The masterchain cannot be split or merged, so a masterchain block usually has exactly one immediate antecessor. The sole exception is the "masterchain block zero", distinguished by having a sequence number equal to zero; it has no antecessors at all, and contains the initial configuration of the whole TON Blockchain (e.g., the original set of validators).

- The masterchain blocks contain another important non-split structure: *ShardHashes*, a binary tree with a list of all defined shardchains along with the hashes of the latest block inside each of the listed shardchains. It is the inclusion of a shardchain block into this structure that makes a shardchain block "canonical", and enables other shardchains' blocks to refer to data (e.g., outbound messages) contained in the shardchain block.

- The state of the masterchain contains global configuration parameters of the whole TON Blockchain, such as the minimum and maximum gas prices, the supported versions of TVM, the minimum stake for the validator candidates, the list of alternative cryptocurrencies supported in addition to Grams, the total amount of Grams issued so far, and the current set of validators responsible for creating and signing new blocks, along with their public keys.

- The state of the masterchain also contains the code of the smart contracts used to elect the subsequent sets of validators and to modify the global configuration parameters. The code of these smart contracts itself is a part of the global configuration parameters and can be modified accordingly. In this respect, this code (along with the current values of these parameters) functions like a "constitution" for the TON Blockchain. It is initially established in masterchain block zero.

- There are no transit messages through the masterchain: each inbound message must have a destination inside the masterchain, and each outbound message must have a source inside the masterchain.

1.3 Consistency conditions

In addition to the data structures contained in the block and in the blockchain state, which are serialized into bags of cells according to certain TL-B schemes

explained in detail later (cf. Chapters **3**–**5**), an important component of the blockchain layout is the *consistency conditions* between data kept inside one or in different blocks (as mentioned in **1.2.3**). This section describes in detail the function of consistency conditions in the blockchain.

1.3.1. Expressing consistency conditions. In principle, dependent data types (such as those used in TL-B) could be used not only to describe the serialization of block data, but also to express conditions imposed on the components of such data types. (For instance, one could define data type *OrderedIntPair*, with pairs of integers (x, y), such that $x < y$, as values.) However, TL-B currently is not expressive enough to encode all the consistency conditions we need, so we opt for a semi-formalized approach in this text. In the future, we may present a subsequent complete formalization in a suitable proof assistant such as Coq.

1.3.2. Importance of consistency conditions. The consistency conditions ultimately are at least as important as the "unrestricted" data structures on which they are imposed, especially in the blockchain context. For instance, the consistency conditions ensure that the state of an account does not change between blocks, and that it can change within a block only as a result of a transaction. In this way, the consistency conditions ensure the safe storage of cryptocurrency balances and other information inside the blockchain.

1.3.3. Kinds of consistency conditions. There are several kinds of consistency conditions imposed on the TON Blockchain:

- *Global conditions* — Express the invariants throughout the entire TON Blockchain. For instance, the *message delivery guarantees*, which assert that each message generated must be delivered to its destination account and delivered exactly once, are part of the global conditions.

- *Internal (local) conditions* — Express the conditions imposed on the data kept inside one block. For example, each transaction included in the block (i.e., present in the transaction list of some account) processes exactly one inbound message; this inbound message must be listed in the *InMsgDescr* structure of the block as well.

- *External (local) conditions* — Express the conditions imposed on the data of different blocks, usually belonging to the same or to neighbor-

ing shardchains (with respect to Hypercube Routing). Therefore, the external conditions come in several flavors:

- *Antecessor/successor conditions* — Express the conditions imposed on the data of some block and of its immediate antecessor or (in the case of a preceding shardchain merge event) two immediate antecessors. The most important of these conditions is the one stating that the initial state for a shardchain block must coincide with final shardchain state of the immediate antecessor block, provided no shardchain split/merge event happened in between.
- *Masterchain/shardchain conditions* — Express the conditions imposed on a shardchain block and on the masterchain block that refers to it in its *ShardHashes* list or is referred to in the header of the shardchain block.
- *Neighbor (block) conditions* — Express the relations between the blocks of neighboring shardchains with respect to Hypercube Routing. The most important of these conditions express the relation between the *InMsgDescr* of a block and the *OutMsgQueue* of the state of a neighboring block.

1.3.4. Decomposition of global and local conditions into simpler local conditions. The *global* consistency conditions, such as the message delivery guarantees, are truly necessary for the blockchain to work properly; however, they are hard to enforce and verify directly. Therefore, we instead introduce a lot of simpler *local* consistency conditions, which are easier to enforce and verify since they involve only one block, or perhaps two adjacent blocks. These local conditions are chosen in such a fashion that the desired global conditions are logical consequences of (the conjunction of) all the local conditions. In this respect, we say that the global conditions have been "decomposed" into simpler local conditions.

Sometimes a local condition still turns out to be too cumbersome to enforce or verify. In that case it is decomposed further, into even simpler local conditions.

1.3.5. Decomposition may require additional data structures and additional internal consistency conditions. The decomposition of a condition into simpler local consistency conditions sometimes requires the introduction of additional data structures. For example, the *InMsgDescr*

explicitly lists all inbound messages processed in a block, even if this list might have been obtained by scanning the list of all the transactions present in the block. However, *InMsgDescr* greatly simplifies the neighbor conditions related to message forwarding and routing, which ultimately add up to the global message delivery guarantees.

Notice that the introduction of such additional data structures is a sort of "database denormalization" (i.e., it leads to some redundancy, or to some data being present more than once), and therefore more internal consistency conditions need to be imposed (e.g., if some data are now present in two copies, we must require that these two copies coincide). For instance, once we introduce *InMsgDescr* to facilitate message forwarding between shardchains, we need to introduce internal consistency conditions relating *InMsgDescr* to the transaction list of the same block.

1.3.6. Correct serialization conditions. Apart from the high-level internal consistency conditions, which treat the contents of a block as a value of an abstract data type, there are some lower-level internal consistency conditions, called "(correct) serialization conditions", which ensure that the tree of cells present in the block is indeed a valid serialization of a value of the expected abstract data type. Such serialization conditions can be automatically generated from the TL-B scheme describing the abstract data type and its serialization into a tree of cells.

Notice that the serialization conditions are a set of mutually recursive predicates on cells or cell slices. For example, if a value of type A consists of a 32-bit magic number m_A, a 64-bit integer l, and two references to cells containing values of types B and C, respectively, then the correct serialization condition for values of type A will require a cell or a cell slice to contain exactly 96 data bits and two cell references r_1 and r_2, with the additional requirements that the first 32 data bits contain m_A, and the two cells referred to by r_1 and r_2 satisfy the serialization conditions for values of types B and C, respectively.

1.3.7. Constructive elimination of existence quantifiers. The local conditions one might want to impose sometimes are *non-constructible*, meaning that they do not necessarily contain an explanation of why they are true. A typical example of such a condition C is given by

$$C :\equiv \forall_{(x:X)} \exists_{(y:Y)} A(x, y) \quad , \tag{5}$$

"for any x from X, there is a y from Y such that condition $A(x, y)$ holds". Even if we know C to be true, we do not have a way of quickly finding a $y : Y$, such that $A(x, y)$, for a given $x : X$. As a consequence, the verification of C may be quite time-consuming.

In order to simplify the verification of local conditions, they are made *constructible* (i.e., verifiable in bounded time) by adding some *witness* data structures. For instance, condition C of (5) may be transformed by adding a new data structure $f : X \to Y$ (a map f from X to Y) and imposing the following condition C' instead:

$$C' :\equiv \forall_{(x:X)} A\big(x, f(x)\big) \quad . \tag{6}$$

Of course, the "witness" value $f(x) : Y$ may be included inside the (modified) data type X instead of being kept in a separate table f.

1.3.8. Example: consistency condition for *InMsgDescr*. For instance, the consistency condition between $X := \mathit{InMsgDescr}$, the list of all inbound messages processed in a block, and $Y := \mathit{Transactions}$, the list of all transactions present in a block, is of the above sort: "For any input message x present in *InMsgDescr*, a transaction y must be present in the block such that y processes x".[8] The procedure of $\exists$-elimination described in **1.3.7** leads us to introduce an additional field in the inbound message descriptors of *InMsgDescr*, containing a reference to the transaction in which the message is actually processed.

1.3.9. Constructive elimination of logical disjunctions. Similarly to the transformation described in **1.3.7**, condition

$$D :\equiv \forall_{(x:X)} \big(A_1(x) \vee A_2(x)\big) \quad , \tag{7}$$

"for all x from X, at least one of $A_1(x)$ and $A_2(x)$ holds", may be transformed into a function $i : X \to \mathbf{2} = \{1, 2\}$ and a new condition

$$D' :\equiv \forall_{(x:X)} A_{i(x)}(x) \tag{8}$$

This is a special case of the existential quantifier elimination considered before for $Y = \mathbf{2} = \{1, 2\}$. It may be useful when $A_1(x)$ and $A_2(x)$ are complicated conditions that cannot be verified quickly, so that it is useful to know in advance which of them is in fact true.

[8] This example is a bit simplified since it does not take into account the presence of transit messages in *InMsgDescr*, which are not processed by any explicit transaction.

For instance, *InMsgDescr*, as considered in **1.3.8**, can contain both messages processed in the block and transit messages. We might introduce a field in the inbound message description to indicate whether the message is transit or not, and, in the latter case, include a witness field for the transaction processing the message.

1.3.10. Constructivization of conditions. This process of eliminating the non-constructible logical binders $\exists$ (existence quantifier) and (sometimes) $\vee$ (logical disjunction) by introducing additional data structures and fields—that is, the process of making a condition constructible—will be called *constructivization*. If taken to its theoretical limit, this process leads to logical formulas containing only universal quantifiers and logical conjunctions, at the expense of adding some witness fields into certain data structures.

1.3.11. Validity conditions for a block. Ultimately, all of the internal conditions for a block, along with the local antecessor and neighbor conditions involving this block and another previously generated block, constitute the *validity conditions* for a shardchain or masterchain block. A block is *valid* if it satisfies the validity conditions. It is the responsibility of validators to generate valid blocks, as well as check the validity of blocks generated by other validators.

1.3.12. Witnesses of the invalidity of a block. If a block does not satisfy all of the validity conditions $C_1, \ldots, C_n$ (i.e., the conjunction $V :\equiv \bigwedge_i C_i$ of the validity conditions), it is *invalid*. This means that it satisfies the "invalidity condition" $\neg V = \bigvee_i \neg C_i$. If all of the C_i—and hence, also V—have been "constructivized" in the sense described in **1.3.10**, so that they contain only logical conjunctions and universal quantifiers (and simple atomic propositions), then $\neg V$ contains only logical disjunctions and existential quantifiers. Then a constructivization of $\neg V$ may be defined, which would involve an *invalidity witness*, starting with an index i of the specific validity condition C_i which fails.

Such invalidity witnesses may also be serialized and presented to other validators or committed into the masterchain to prove that a specific block or block candidate is in fact invalid. Therefore, the construction and serialization of invalidity witnesses is an important part of a Proof-of-Stake (PoS) blockchain design.[9]

[9]It is interesting to note that this part of the work can be done almost automatically.

1.3.13. Minimizing the size of witnesses. An important consideration for the design of the local conditions, their decomposition into simpler conditions, and their constructivization is to make the verification of each condition as simple as possible. However, another requirement is that we should minimize the size of witnesses both for a condition (so that block size does not grow too much during the constructivization process) and for its negation (so that the invalidity proofs have bounded size, which simplifies their verification, transmission, and inclusion into the masterchain). These two design principles are sometimes at odds, and a compromise must be then sought.

1.3.14. Minimizing the size of Merkle proofs. The consistency conditions are originally intended to be processed by a party who already has all the relevant data (e.g., all the blocks mentioned in the condition). On some occasions, however, they must be verified by a party who does not have all the blocks in question, but knows only their hashes. For example, suppose that a block invalidity proof were augmented by the signature of a validator that had signed an invalid block (and therefore would have to be punished). In this case, the signature would contain only the hash of the wrongly signed block; the block itself would have to be recovered from a different place before verifying the block invalidity proof.

A compromise between providing only the hash of the supposedly invalid block and providing the entire invalid block along with the invalidity witness is to augment the invalidity witness by a Merkle proof starting from the hash of the block (i.e., of the root cell of the block). Such a proof would include all the cells referred to in the invalidity witness, along with all the cells on the paths from these cells to the root cells and the hashes of their siblings. Then an invalidity proof becomes self-contained enough to provide sufficient justification on its own for punishing a validator. For example, the invalidity proof suggested above might be presented to a smart contract residing in the masterchain that punishes the validators for incorrect behavior.

Since such an invalidity proof must be augmented by a Merkle proof, it makes sense to write the consistency conditions so that the Merkle proofs for their negations would be as small as possible. In particular, each individual condition must be as "local" as possible (i.e., involve a minimal number of cells). This also optimizes the verification time of the invalidity proof.

1.3.15. Collated data for the external conditions. When a validator suggests an unsigned block to the other validators of a shardchain, these other validators must check the validity of this block candidate—i.e., verify

that it satisfies all of the internal and external local consistency conditions. While the internal conditions do not require any extra data in addition to the block candidate itself, the external conditions need some other blocks, or at least some information out of those blocks. Such additional information may be extracted from those blocks, along with all cells on the paths from the cells containing the required additional information to the root cell of the corresponding blocks and the hashes of the siblings of the cells on these paths, to present a Merkle proof that can be processed without knowledge of the referred blocks themselves.

This additional information, called *collated data*, is serialized as a bag of cells and presented by the validator along with the unsigned block candidate itself. The block candidate along with the collated data is called a *collated block*.

1.3.16. Conditions for a collated block. The *external* consistency conditions for a block candidate are thus (automatically) transformed into *internal* consistency conditions for a collated block, which greatly simplifies and speeds up their verification by the other validators. However, some data—such as the final state of the immediate antecessor of the block being validated—is not collated. Instead, all validators are supposed to keep a local copy of this data.

1.3.17. Representation conditions and representation hashes. Notice that once Merkle proofs are included into a collated block, the consistency conditions must take into account which data (i.e., which cells) are actually present in the collated block, and not just referred to by their hashes. This leads to a new group of conditions, called *representation conditions*, which must be able to distinguish an external cell reference (usually represented by its 256-bit hash) from the cell itself. A validator can be punished for suggesting a collated block that does not contain all of the expected collated data inside, even if the block candidate itself is valid.

This also leads to the utilization of *representation hashes* instead of *transparent hashes* for collated blocks.

1.3.18. Verification in the absence of the collated data. Notice that a block must still be verifiable in the absence of the collated data; otherwise, no party except the validators would be able to check a previously committed block by its own means. In particular, witnesses cannot be included into the collated data: they must reside in the block itself. The collated data

must contain only some portions of neighboring blocks referred to in the principal block along with suitable Merkle proofs, which can be reconstructed by anybody who has the referenced blocks themselves.

1.3.19. Inclusion of Merkle proofs in the block itself. Notice that on some occasions Merkle proofs must be embedded into the block itself, and not just into collated data. For instance:

- During Instant Hypercube Routing (IHR), a message may be included directly into the *InMsgDescr* of a block of the destination shardchain, without travelling all the way along the edges of the hypercube. In this case, a Merkle proof of the existence of the message in the *OutMsgDescr* of a block of the originating shardchain must be included into *InMsgDescr* along with the message itself.

- An invalidity proof, or another proof of validator misbehavior, may be committed into the masterchain by including it in the body of a message sent to a special smart contract. In this case, the invalidity proof must include some cells along with a Merkle proof, which must therefore be contained in a message body.

- Similarly, a smart contract defining a payment channel, or another kind of side-chain, may accept finalization messages or misbehavior proof messages that contain suitable Merkle proofs.

- The final state of a shardchain is not included into a shardchain block. Instead, only the cells that have been modified are included; those cells that are inherited from the old state are referred to by their hashes, along with suitable Merkle proofs consisting of the cells on the path from the root of the old state to the cells of the old state referred to.

1.3.20. Provisions for handling incomplete data. As we have seen, it is necessary to include incomplete data and Merkle proofs into the body of a block, into the body of some messages contained in a block, and into the state. This necessity is reflected by some extra representation conditions, as well as provisions for the messages (and by extension, the cell trees processed by TVM) to contain incomplete data (external cell references and Merkle proofs). In most cases, such external cell references contain only the 256-bit SHA256 hash of a cell along with a flag; if a smart contract attempts to inspect the contents of such a cell by a `CTOS` primitive (e.g., for deserialization), an

exception is triggered. However, an external reference to such a cell can be stored into the smart contract's persistent storage, and both the transparent and the representation hashes of such a cell can be computed.

1.4 Logical time and logical time intervals

This section takes a closer look at so-called *logical time*, extensively used in the TON Blockchain for message forwarding and message delivery guarantees, among other purposes.

1.4.1. Logical time. A component of the TON Blockchain that also plays an important role in message delivery is the *logical time*, usually denoted by Lt. It is a non-negative 64-bit integer, assigned to certain events roughly as follows:

> If an event e logically depends on events $e_1, \ldots, e_n$, then $\mathrm{Lt}(e)$ is the smallest non-negative integer greater than all $\mathrm{Lt}(e_i)$.

In particular, if $n = 0$ (i.e., if e does not depend on any prior events), then $\mathrm{Lt}(e) = 0$.

1.4.2. A relaxed variant of logical time. On some occasions we relax the definition of logical time, requesting only that

$$\mathrm{Lt}(e) > \mathrm{Lt}(e') \quad \text{whenever } e \succ e' \text{ (i.e., } e \text{ logically depends on } e'), \tag{9}$$

without insisting that $\mathrm{Lt}(e)$ be the smallest non-negative integer with this property. In such cases we can speak about *relaxed* logical time, as opposed to the *strict* logical time defined above (cf. **1.4.1**). Notice, however, that the condition (9) is a fundamental property of logical time and cannot be relaxed further.

1.4.3. Logical time intervals. It makes sense to assign to some events or collections of events C an *interval* of logical times $\mathrm{Lt}^{\bullet}(C) = [\mathrm{Lt}^{-}(C), \mathrm{Lt}^{+}(C))$, meaning that the collection of events C took place in the specified "interval" of logical times, where $\mathrm{Lt}^{-}(C) < \mathrm{Lt}^{+}(C)$ are some integers (64-bit integers in practice). In this case, we can say that C *begins* at logical time $\mathrm{Lt}^{-}(C)$, and *ends* at logical time $\mathrm{Lt}^{+}(C)$.

By default, we assume $\mathrm{Lt}^{+}(e) = \mathrm{Lt}(e)+1$ and $\mathrm{Lt}^{-}(e) = \mathrm{Lt}(e)$ for simple or "atomic" events, assuming that they last exactly one unit of logical time.

In general, if we have a single value $\mathrm{LT}(C)$ as well as logical time interval $\mathrm{LT}^\bullet(C) = [\mathrm{LT}^-(C), \mathrm{LT}^+(C))$, we always require that

$$\mathrm{LT}(C) \in [\mathrm{LT}^-(C), \mathrm{LT}^+(C)) \tag{10}$$

or, equivalently,

$$\mathrm{LT}^-(C) \le \mathrm{LT}(C) < \mathrm{LT}^+(C) \tag{11}$$

In most cases, we choose $\mathrm{LT}(C) = \mathrm{LT}^-(C)$.

1.4.4. Requirements for logical time intervals. The three principal requirements for logical time intervals are:

- $0 \le \mathrm{LT}^-(C) < \mathrm{LT}^+(C)$ are non-negative integers for any collection of events C.
- If $e' \prec e$ (i.e., if an atomic event e logically depends on another atomic event e'), then $\mathrm{LT}^\bullet(e') < \mathrm{LT}^\bullet(e)$ (i.e., $\mathrm{LT}^+(e') \le \mathrm{LT}^-(e)$).
- If $C \supset D$ (i.e., if a collection of events C contains another collection of events D), then $\mathrm{LT}^\bullet(C) \supset \mathrm{LT}^\bullet(D)$, i.e.,

$$\mathrm{LT}^-(C) \le \mathrm{LT}^-(D) < \mathrm{LT}^+(D) \le \mathrm{LT}^+(C) \tag{12}$$

  In particular, if C consists of atomic events $e_1, \dots, e_n$, then $\mathrm{LT}^-(C) \le \inf_i \mathrm{LT}^-(e_i) \le \inf_i \mathrm{LT}(e_i)$ and $\mathrm{LT}^+(C) \ge \sup_i \mathrm{LT}^+(e_i) \ge 1 + \sup_i \mathrm{LT}(e_i)$.

1.4.5. Strict, or minimal, logical time intervals. One can assign to any finite collection of atomic events $E = \{e\}$ related by a causality relation (partial order) $\prec$, and all subsets $C \subset E$, *minimal* logical time intervals. That is, among all assignments of logical time intervals satisfying the conditions listed in **1.4.4**, we choose the one having all $\mathrm{LT}^+(C) - \mathrm{LT}^-(C)$ as small as possible, and if several assignments with this property exist, we choose the one that has the minimum $\mathrm{LT}^-(C)$ as well.

Such an assignment can be achieved by first assigning logical time $\mathrm{LT}(e)$ to all atomic events $e \in E$ as described in **1.4.1**, then setting $\mathrm{LT}^-(C) := \inf_{e \in C} \mathrm{LT}(e)$ and $\mathrm{LT}^+(C) := 1 + \sup_{e \in C} \mathrm{LT}(e)$ for any $C \subset E$.

In most cases when we need to assign logical time intervals, we use the minimal logical time intervals just described.

1.4.6. Logical time in the TON Blockchain. The TON Blockchain assigns logical time and logical time intervals to several of its components.

For instance, each outbound message created in a transaction is assigned its *logical creation time*; for this purpose, the creation of an outbound message is considered an atomic event, logically dependent on the previous message created by the same transaction, as well as on the previous transaction of the same account, on the inbound message processed by the same transaction, and on all events contained in the blocks referred to by hashes contained in the block with the same transaction. As a consequence, *outbound messages created by the same smart contract have strictly increasing logical creation times.* The transaction itself is considered a collection of atomic events, and is assigned a logical time interval (cf. **4.2.1** for a more precise description).

Each block is a collection of transaction and message creation events, so it is assigned a logical time interval, explicitly mentioned in the header of the block.

1.5 Total blockchain state

This section discusses the total state of the TON Blockchain, as well as the states of separate shardchains and the masterchain. For example, the precise definition of the state of the neighboring shardchains becomes crucial for correctly formalizing the consistency condition asserting that the validators for a shardchain must import the oldest messages from the union of *OutMsgQueue*s taken from the states of all neighboring shardchains (cf. **2.2.5**).

1.5.1. Total state defined by a masterchain block. Every masterchain block contains a list of all currently active shards and of the latest blocks for each of them. In this respect, *every masterchain block defines the corresponding total state of the TON Blockchain, since it fixes the state of every shardchain, and of the masterchain as well.*

An important requirement imposed on this list of the latest blocks for all shardchain blocks is that, if a masterchain block B lists S as the latest block of some shardchain, and a newer masterchain block B', with B as one of its antecessors, lists S' as the latest block of the same shardchain, then S must be one of the antecessors of S'.[10] This condition makes the total state of the

[10]In order to express this condition correctly in the presence of dynamic sharding, one should fix some account ξ, and consider the latest blocks S and S' of the shardchains containing ξ in the shard configurations of both B and B', since the shards containing ξ

TON blockchain defined by a subsequent masterchain block B' compatible with the total state defined by a previous block B.

1.5.2. Total state defined to by a shardchain block. Every shardchain block contains the hash of the most recent masterchain block in its header. Consequently, all the blocks referred to in that masterchain block, along with their antecessors, are considered "known" or "visible" to the shardchain block, and no other blocks are visible to it, with the sole exception of its antecessors inside its proper shardchain.

In particular, when we say that a block *must* import in its *InMsgDescr* the messages from the *OutMsgQueue* of the states of all neighboring shardchains, it means that precisely the blocks of other shardchains visible to that block must be taken into account, and at the same time the block cannot contain messages from "invisible" blocks, even if they are otherwise correct.

1.6 Configurable parameters and smart contracts

Recall that the TON Blockchain has several so-called "configurable parameters" (cf. [3]), which are either certain values or certain smart contracts residing in the masterchain. This section discusses the storage of and access to these configurable parameters.

1.6.1. Examples of configurable parameters. The properties of the blockchain controlled by configurable parameters include:

- The minimum stake for validators.
- The maximum size of the group of elected validators.
- The maximum number of blocks for which the same group of validators are responsible.
- The validator election process.
- The validator punishing process.
- The currently active and the next elected set of validators.

might be different in B and B'.

- The process of changing configurable parameters, and the address of the smart contract γ responsible for holding the values of the configurable parameters and for modifying their values.

1.6.2. Location of the values of configurable parameters. The configurable parameters are kept in the persistent data of a special configuration smart contract γ residing in the masterchain of the TON Blockchain. More precisely, the first reference of the root cell of the persistent data of that smart contract is a dictionary mapping 64-bit keys (parameter numbers) to the values of the corresponding parameters; each value is serialized into a cell slice according to the type of that value. If a value is a "smart contract" (necessarily residing in the masterchain), its 256-bit account address is used instead.

1.6.3. Quick access through the header of masterchain blocks. To simplify access to the current values of configurable parameters, and to shorten the Merkle proofs containing references to them, the header of each masterchain block contains the address of smart contract γ. It also contains a direct cell reference to the dictionary containing all values of configurable parameters, which lies in the persistent data of γ. Additional consistency conditions ensure that this reference coincides with the one obtained by inspecting the final state of smart contract γ.

1.6.4. Getting values of configurable parameters by get methods. The configuration smart contract γ provides access to some of configurable parameters by means of "get methods". These special methods of the smart contract do not change its state, but instead return required data in the TVM stack.

1.6.5. Getting values of configurable parameters by get messages. Similarly, the configuration smart contract γ may define some "ordinary" methods (i.e., special inbound messages) to request the values of certain configuration parameters, which will be sent in the outbound messages generated by the transaction processing such an inbound message. This may be useful for some other fundamental smart contracts that need to know the values of certain configuration parameters.

1.6.6. Values obtained by get methods may be different from those obtained through the block header. Notice that the state of the configuration smart contract γ, including the values of configurable parameters,

may change several times inside a masterchain block, if there are several transactions processed by γ in that block. As a consequence, the values obtained by invoking get methods of γ, or sending get messages to γ, may be different from those obtained by inspecting the reference in the block header (cf. **1.6.3**), which refers to the *final* state of the configurable parameters in the block.

1.6.7. Changing the values of configurable parameters. The procedure for changing the values of configurable parameters is defined in the code of smart contract γ. For most configurable parameters, called *ordinary*, any validator may suggest a new value by sending a special message with the number of the parameter and its proposed value to γ. If the suggested value is valid, further voting messages from the validators are collected by the smart contract, and if more than two-thirds each of the current and next sets of validators support the proposal, the value is changed.

Some parameters, such as the current set of validators, cannot be changed in this way. Instead, the current configuration contains a parameter with the address of smart contract ν responsible for electing the next set of validators, and smart contract γ accepts messages only from this smart contract ν to modify the value of the configuration parameter containing the current set of validators.

1.6.8. Changing the validator election procedure. If the validator election procedure ever needs to be changed, this can be accomplished by first committing a new validator election smart contract into the masterchain, and then changing the ordinary configurable parameter containing the address ν of the validator election smart contract. This will require two-thirds of the validators to accept the proposal in a vote as described above in **1.6.7**.

1.6.9. Changing the procedure of changing configurable parameters. Similarly, the address of the configuration smart contract itself is a configurable parameter and may be changed in this fashion. In this way, most fundamental parameters and smart contracts of the TON Blockchain may be modified in any direction agreed upon by the qualified majority of the validators.

1.6.10. Initial values of the configurable parameters. The initial values of most configurable parameters appear in block zero of the masterchain as part of the masterchain's initial state, which is explicitly present with no

omissions in this block. The code of all fundamental smart contracts is also present in the initial state. In this way, the original "constitution" and configuration of the TON Blockchain, including the original set of validators, is made explicit in block zero.

1.7 New smart contracts and their addresses

This section discusses the creation and initialization of new smart contracts—in particular, the origin of their initial code, persistent data, and balance. It also discusses the assignment of account addresses to new smart contracts.

1.7.1. Description valid only for masterchain and basic workchain. The mechanisms for creating new smart contracts and assigning their addresses described in this section are valid only for the basic workchain and the masterchain. Other workchains may define their own mechanisms for dealing with these problems.

1.7.2. Transferring cryptocurrency to uninitialized accounts. First of all, *it is possible to send messages, including value-bearing messages, to previously unmentioned accounts.* If an inbound message arrives at a shardchain with a destination address η corresponding to an undefined account, it is processed by a transaction as if the code of the smart contract were empty (i.e., consisting of an implicit `RET`). If the message is value-bearing, this leads to the creation of an "uninitialized account", which may have a non-zero balance (if value-bearing messages have been sent to it),[11] but has no code and no data. Because even an uninitialized account occupies some persistent storage (needed to hold its balance), some small persistent-storage payments will be exacted from time to time from the account's balance, until it becomes negative.

1.7.3. Initializing smart contracts by constructor messages. An account, or smart contract, is created by sending a special *constructor message* M to its address η. The body of such a message contains the tree of cells with the initial code of the smart contract (which may be replaced by its hash in some situations), and the initial data of the smart contract (maybe empty; it can be replaced by its hash). The hash of the code and of the data

[11] Value-bearing messages with the `bounce` flag set will not be accepted by an uninitialized account, but will be "bounced" back.

contained in the constructor message must coincide with the address η of the smart contract; otherwise, it is rejected.

After the code and data of the smart contract are initialized from the body of the constructor message, the remainder of the constructor message is processed by a transaction (the *creating transaction* for smart contract η) by invoking TVM in a manner similar to that used for processing ordinary inbound messages.

1.7.4. Initial balance of a smart contract. Notice that the constructor message usually must bear some value, which will be transferred to the balance of the newly-created smart contract; otherwise, the new smart contract would have a balance of zero and would not be able to pay for storing its code and data in the blockchain. The minimum balance required from a newly-created smart contract is a linear (more precisely, affine) function of the storage it uses. The coefficients of this function may depend on the workchain; in particular, they are higher in the masterchain than in the basic workchain.

1.7.5. Creating smart contracts by external constructor messages. In some cases, it is necessary to create a smart contract by a constructor message that cannot bear any value—for instance, by a constructor message "from nowhere" (an external inbound message). Then one should first transfer a sufficient amount of funds to the uninitialized smart contract as explained in **1.7.2**, and only then send a constructor message "from nowhere".

1.7.6. Example: creating a cryptocurrency wallet smart contract. An example of the above situation is provided by cryptocurrency wallet applications for human users, which must create a special wallet smart contract in the blockchain in which to keep the user's funds. This can be achieved as follows:

- The cryptocurrency wallet application generates a new cryptographic public/private key pair (typically for Ed25519 elliptic curve cryptography, supported by special TVM primitives) for signing the user's future transactions.

- The cryptocurrency wallet application knows the code of the smart contract to be created (which typically is the same for all users), as well as the data, which typically consists of the public key of the wallet

(or of its hash) and is generated at the very beginning. The hash of this information is the address ξ of the wallet smart contract to be created.

- The wallet application may display the user's address ξ, and the user may start to receive funds to her uninitialized account ξ—for example, by buying some cryptocurrency at an exchange, or by asking a friend to transfer a small sum.

- The wallet application can inspect the shardchain containing account ξ (in the case of a basic workchain account) or the masterchain (in the case of a masterchain account), either by itself or using a blockchain explorer, and check the balance of ξ.

- If the balance is sufficient, the wallet application may create and sign (with the user's private key) the constructor message ("from nowhere"), and submit it for inclusion to the validators or the collators for the corresponding blockchain.

- Once the constructor message is included into a block of the blockchain and processed by a transaction, the wallet smart contract is finally created.

- When the user wants to transfer some funds to some other user or smart contract η, or wants to send a value-bearing message to η, she uses her wallet application to create the message m that she wants her wallet smart contract ξ to send to η, envelope m into a special "message from nowhere" m' with destination ξ, and sign m' with her private key. Some provisions against replay attacks must be made, as explained in **2.2.1**.

- The wallet smart contract receives message m' and checks the validity of the signature with the aid of the public key stored in its persistent data. If the signature is correct, it extracts embedded message m from m' and sends it to its intended destination η, with the indicated amount of funds attached to it.

- If the user does not need to immediately start transferring funds, but only wants to passively receive some funds, she may keep her account uninitialized as long as she wants (provided the persistent storage payments do not lead to the exhaustion of its balance), thus minimizing the storage profile and persistent storage payments of the account.

- Notice that the wallet application may create for the human user the illusion that the funds are kept in the application itself, and provide an interface to transfer funds or send arbitrary messages "directly" from the user's account ξ. In reality, all these operations will be performed by the user's wallet smart contract, which effectively acts as a proxy for such requests. We see that a cryptocurrency wallet is a simple example of a *mixed* application, having an on-chain part (the wallet smart contract, used as a proxy for outbound messages) and an off-chain part (the external wallet application running on a user's device and keeping the private account key).

Of course, this is just one way of dealing with the simplest user wallet smart contracts. One can create multi-signature wallet smart contracts, or create a shared wallet with internal balances kept inside it for each of its individual users, and so on.

1.7.7. Smart contracts may be created by other smart contracts. Notice that a smart contract may generate and send a constructor message while processing any transaction. In this way, smart contracts may automatically create new smart contracts, if they need to, without any human intervention.

1.7.8. Smart contracts may be created by wallet smart contracts. On the other hand, a user may compile the code for her new smart contract ν, generate the corresponding constructor message m, and use the wallet application to force her wallet smart contract ξ to send message m to ν with an adequate amount of funds, thus creating the new smart contract ν.

1.8 Modification and removal of smart contracts

This section explains how the code and state of a smart contract may be changed, and how and when a smart contract may be destroyed.

1.8.1. Modification of the data of a smart contract. The persistent data of a smart contract is usually modified as a result of executing the code of the smart contract in TVM while processing a transaction, triggered by an inbound message to the smart contract. More specifically, the code of the smart contract has access to the old persistent storage of the smart contract via TVM control register `c4`, and may modify the persistent storage by storing another value into `c4` before normal termination.